

# Relational Constraint Driven Test Case Synthesis for Web Applications

Xiang Fu

Hofstra University  
Hempstead, NY 11549

Xiang.Fu@hofstra.edu

This paper proposes a relational constraint driven technique that synthesizes test cases automatically for web applications. Using a static analysis, servlets can be modeled as relational transducers, which manipulate backend databases. We present a synthesis algorithm that generates a sequence of HTTP requests for simulating a user session. The algorithm relies on backward symbolic image computation for reaching a certain database state, given a code coverage objective. With a slight adaptation, the technique can be used for discovering workflow attacks on web applications.

## 1 Introduction

Modern web applications usually rely on backend database systems for storing important system information or supporting business decisions. The complexity of database queries, however, often complicates the task of thoroughly testing a web application. To manually design test cases involves labor intensive initialization of database systems, even with the help of unit testing tools such as SQLUnit [25] and DBUnit [8]. It is desirable to automatically synthesize test cases for web applications.

There has been a strong interest recently in testing database driven applications and database management systems (see e.g., [12, 4, 20]). Many of them are query aware, i.e., given a SQL query, an initial database (DB) instance is generated to make that query satisfiable. The DB instance is fed to the target web application as input, so that a certain code coverage goal is achieved. The problem we are trying to tackle is one step further – it is a *synthesis problem*: given a certain database state (or a relational constraint), a *call sequence* of web servlets is synthesized to reach the given DB state. This is motivated by the special architecture of web applications. Unlike typical desktop software systems, the atomic components of a web application, i.e., *web servlets*, are accessible to end users. In addition, cookies and session variables are frequently used for session maintenance and user tracing. Thus web application testing is usually session oriented (see, e.g., [22, 11]). Unlike unit testing general database driven applications, an intermediate database state has to be resulted from a call sequence and cannot be initialized at will. We are interested in synthesizing such a call sequence. The technique could also be leveraged to detect workflow attacks [3, 7], where an “unexpected” call sequence can cause harm to the business logic of a web application (e.g., shipping a product without charging credit card).

We propose a white-box analysis, which consists of the following steps: (1) *interface extraction*: each web servlet is modeled as a collection of *path transducers*. A path transducer is essentially a relational transducer [2] that corresponds to a single execution path of the web servlet. A transducer is represented as a pair of pre/post conditions, built upon relational constraints. Path transducers are extracted using light-weight symbolic execution technique. (2) *coverage goal generation*: to achieve a certain coverage goal, symbolic execution is used to generate relational constraints (expressed using first order relational logic [18]). (3) *call sequence synthesis*: a heuristic algorithm is used to determine a call sequence that

could lead to the required intermediate database state. Symbolic constraint solving is currently used for performing backward/forward symbolic image computation of transducers. Best effort constraint solving (restricted to finite model and finite scope), based on Alloy Analyzer [17], is used to generate parameters of each HTTP request in the call sequence.

Although there does not exist an implementation for the proposed technique, we plan to illustrate its effectiveness and feasibility using a case study. §2 presents a motivating example. §3 introduces the path transducer model. §4 discusses symbolic image computation of relational algebra. The call sequence algorithm is presented in §5. §6 adapts the algorithm for discovering workflow attacks. §7 discusses related work and concludes.

## 2 Motivating Example

This section introduces SimpleScarf, a case study example used throughout the paper. SimpleScarf is adapted from the Scarf conference management system [26]. It is comprised of five servlets for managing the membership of a conference. Each servlet is implemented as a PHP file that accepts HTML requests and generates HTML responses.

These servlets are briefly described as follows. Later we will present the formal relational transducer model for each of them. (1) `Showsessions.php` displays the list of paper sessions that are associated with the current user. (2) `Insertsession.php` adds a new paper session to the system. (3) `Addmember.php` inserts a new member into a paper session. (4) `Generaloptions.php` creates a new user of the system. (5) `Login.php` authenticates the log-in process. Once a user successfully logs in, the servlet sets a session variable called “user” for tracking the user session.

There is a backend database with three relations: (1) `Users(uname, enc_wd)` contains information about a user: the user name and the encrypted password; (2) `Sessions(sid, sname)` has two attributes: the paper session id and session name; and (3) `Member(sid, uname)` describes the members of a paper session. To simplify the scenario, the data type of each column is `varchar`. For relation `Member`, there are two foreign key dependency constraints: (1) `sid` on `Sessions`, and (2) `uname` on `Users`. Throughout the paper, each relation or servlet is denoted using the first character of its name, e.g., `Showsessions.php` denoted by `S` and `Users` denoted by `U`.

As an example, Listing 1 presents a fragment of `Showsessions.php`. The servlet consists of two parts: (1) a branch statement which examines a user’s login status, by checking the value of session variable “uname”; and (2) a loop that reads the information retrieved by a `SELECT` statement to generate a list of paper sessions related to the user.

**Coverage Goal:** line 45 in Listing 1. We are interested in two questions: (1) what is the database state or relational constraint that leads the execution of `Showsessions.php` to line 45? (2) what is the call sequence that generates the desired database state? Later, we will show that an algorithm can automatically synthesize a call sequence IGALS and the HTML parameters in each HTTP request.

## 3 Path Transducer Model

We intend to model each *execution path* of a servlet, as an *atomic relational transducer*. For example, one such execution path of `Showsessions.php` could be 1-2-3-4-6-7-8-12-13-15-16-44-45-44-90 (line numbers). We trace such an execution using a “path condition”, i.e., the conjunction of branch conditions

```

1<?php
2 include_once("header.php");
3 print "<div style='float:right'>";
4 if (!isset($_SESSION['uname']))
5     print "<a...> Not logged in </a><br>";
6 else
7     print "Logged in as " . $_SESSION['uname'] . " .";
8 print "</div><br>";
12 $result = query("SELECT session_id...");
13 if (mysql_num_rows($result) == 0) {
14     print ("Sorry, ...");
15 }
16 $curday = -1;
44 while ($row = mysql_fetch_array($result)) {
45     $day = date("F-j", $row['starttime']);
    ...
90}

```

Listing 1: Fragment of Showsessions.php

along the execution path. Note that each servlet may have an infinite set of such transducers. During a static analysis, we can bound loop iterations and recursive call depth for achieving a finite set.

Because each transducer only models one path, it is essentially a transition rule that is expressed as a Boolean combination of relational expressions. The first order relational logic [1] is suitable for expressing uninterpreted functions and next state variables and relations. Here a primed variable (i.e., with a single quote) represents the value of the variable in the next state. Input parameters are preceded with a dollar sign, e.g.,  $\$u, \$p, \$s$  represent the input parameters `uname`, `password`, `session` name in a HTTP request. Session variables are denoted using  $\#$ , e.g.,  $\#u$  represents the session variable `uname`. If in a transition rule, a session variable or relation does not have the primed form, then its value remains the same after the execution of the transition. Relations are generally represented as sets of tuples.

**Definition 3.1** A path transducer is a tuple  $(\mathcal{D}, \text{Dom}, \vec{V}, \vec{A})$  where  $\mathcal{D}$  is the data schema (a finite set of relation schema),  $\text{Dom}$  is an infinite but countable domain for  $\mathcal{D}$ ,  $\vec{V}$  is a finite set of session variables (letting  $\vec{V}'$  be the set of primed variables), and  $\vec{A}$  is a Boolean combination of terms. Each term has one of the following forms:

1. (equality)  $u = E(\vec{v}, \text{Dom})$ , where  $u \in \vec{V} \cup \vec{V}'$  and  $E$  is an expression on the current state of variables and constants from  $\text{Dom}$ .
2. (satisfiability check)  $\text{SAT}(\Psi)$  where  $\Psi$  is a relational algebra formula on  $\mathcal{D}$ .

Notice that in the equality form, a primed variable (next state) is allowed to appear on the LHS (left hand side) only. The syntax (expressiveness) of  $E$  is affected by the decision procedure used in analysis. Currently, we allow Presburger arithmetic [19] and Simple Linear String Equation [13]. Uninterpreted functions are allowed with first order relational logic on  $\mathcal{D}$ , but not with Presburger arithmetic. A relational constraint is defined similarly as a transducer. It is a Boolean combination of equality and satisfiability terms, except that no primed variable occurs. It is formally defined as below.

**Definition 3.2** Let  $\mathcal{A}$  be the set of relational algebra formulae over  $\mathcal{D}$ . Let  $\phi_1, \phi_2 \in \mathcal{A}$  then all of the following also belong to  $\mathcal{A}$ : (1) (selection)  $\sigma_{i=x} \phi_1$  where  $i \in N$  and  $x \in \vec{V} \cup \text{Dom}$ ; (2) (projection)  $\pi_{i_1, i_2, \dots, i_k} \phi_1$  where  $i_1, i_2, \dots, i_k \in N$ ; (3) (cross-product)  $\phi_1 \times \phi_2$ ; (4) (union)  $\phi_1 \cup \phi_2$ ; and (5) (difference)  $\phi_1 - \phi_2$ .

Var	Type	Meaning	Var	Type	Meaning
$\#u$	Session Var	User Name	$\$s_I$	HTTP Param	session name for Insertsessions.php
$\$s_A$	HTTP Param	session name for Addmember.php	$\$u_A$	HTTP Param	user name for Addmember.php
$\$p_G$	HTTP Param	password for Generaloptions.php	$\$u_G$	HTTP Param	uname for Generaloptions.php
$\$p_L$	HTTP Param	password for Login.php	$\$u_L$	HTTP Param	uname for Login.php
$f$	HTTP Param	encryption function	$r$	HTTP Param	password safety constraint
$U$	Relation	Users	$S$	Relation	Sessions
$M$	Relation	Members			

Figure 1: Table of Notations

We now list one sample path transducer for each of the five servlets. We first describe the function of a servlet and then formally define it as a transition rule. Figure 1 summarizes the semantics of all notations to be used in the formulae.

**Showsessions.php:** The servlet first checks the existence of a session variable “uname” (represented by  $\#u$ ), and then verifies if there is at least one session of which  $\#u$  is a member. If the condition evaluates to true, the servlet executes a loop that displays the session name by retrieving information from table Sessions ( $S$ ). Because the loop does not have any side effects (e.g., updating databases), the operations on  $S$  are not included in the path condition during symbolic execution. The following constraint is used to represent the transducer.

$$\#u \neq \text{null} \wedge \text{SAT}(\sigma_{2=\#u}M) \quad (1)$$

Here SAT tests if the input (a relational algebra formula) is satisfiable. For convenience, we omitted the formula  $\#u' = \#u \wedge S' = S \wedge M' = M \wedge U' = U$  (i.e., maintaining the values of all variables and relations), but it needs to be encoded in implementation.

**Insertsessions.php:** The servlet inserts a new session record into relation  $S$ . It accepts one input parameter  $\$s_I$  (the session name). The primary key `sid` is automatically incremented by the system. The action is modeled using the standard set union operator. It is also an example of integer constraints involved in the application.

$$S' = S \cup \{(|S| + 1, \$s_I)\} \quad (2)$$

**Addmember.php:** The servlet takes two parameters: user name ( $\$u_A$ ) and session name ( $\$s_A$ ). Then it looks up for the corresponding session id, and then inserts a tuple into relation  $M$  (Membership). Here we assume that the prime key constraint is valid (every record in  $S$  has a distinct `sid`). So at any time, at most one new record is inserted into  $M$ .

$$M' = M \cup \pi_1(\sigma_{2=\$s_A}(S)) \times \{(\$u_A)\} \quad (3)$$

**Generaloptions.php:** The transducer adds a user to the system. It takes two parameters: user name ( $\$u_G$ ) and password ( $\$p_G$ ).  $f$  is an uninterpreted function that represents the encryption procedure. All uninterpreted functions are assumed to be deterministic.

$$U' = U \cup \{(\$u_G, f(\$p_G))\} \wedge r(\$p_G) \quad (4)$$

In practice, there are security conditions that a legitimate user name and password must meet. This is represented using function  $r(\$p_G)$  where  $r(x)$  is a Boolean function  $\Sigma^* \rightarrow \{T, F\}$ . For example, we

can define it as  $r(x) = x.\text{contains}(\text{"\#"}) \wedge |x| > 7$ . The solution of string constraints can be handled by string analysis (see e.g., [13]).

**Login.php:** The servlet takes two parameters: `uname` ( $\$u_L$ ) and `password` ( $\$p_L$ ), and then verifies their existence in the database. It updates the session variable `#u` with the value of  $\$u_L$ , so that the user session can now be traced by `#u`. The servlet assumes that `#u` is not set before the login activity. Also note that  $(\$u_L, f(\$p_L)) \in U$  is a syntactic sugar for  $\text{SAT}(\sigma_1=\$u_L \sigma_2=f(\$p_L)(U))$ .

$$(\$u_L, f(\$p_L)) \in U \wedge \#u' = \$u_L \wedge \#u = \text{null} \quad (5)$$

### 3.1 Discussions

We outline the idea of extracting path transducer models, though an implementation does not exist yet. Symbolic execution [21] is the major technical approach, and Halfond’s recent work [16] in interface extraction can be leveraged. A servlet is treated as a program, which takes global input (the GET and POST variables). These global variables are replaced by symbolic literals. When an assignment occurs, the variable on LHS (left hand side) is associated with a symbolic expression. Then, at a branch decision, the jump to one of the branches is made randomly (so that it guarantees that each branch will be covered). When entering a branch, the corresponding condition is joined with the current *path condition*. Clearly, the path condition records the conjunction of all conditions that the initial input has to meet to reach the current execution location. Some known system calls (e.g., those to backend databases) are intercepted and replaced with the corresponding symbolic expression. Others (unhandled) are abstracted as uninterpreted functions. Eventually when the symbolic execution completes, the path transducer is the conjunction of three components: (1) the path condition, (2) assignments that change values of global session variables, and (3) operations that update the contents of database relations. String analysis plays an important role in deciding the precision of the translation from symbolic string expressions to relational algebra formulae. Given a string expression (consisting of constant words and string variables) that represents a SQL query, populating the string variables with “benign” values generates a real SQL query which can be parsed to a SQL syntax tree and then be translated into relational algebra formulae (with variables reloaded). Note that here we ignore SQL injection attacks.

## 4 Relational Constraint

This section introduces the preliminary results about relational constraints. Inspired by the work on testing DBMS by Khurshid *et al.* [20], we translate a relational algebra formula to a relational logic formula and then use Alloy Analyzer [17] to find a model in a finite scope. Our initial experiments show that primary and foreign key constraints can be conveniently modeled. Within a small scope, Alloy can quickly find solutions that satisfy a relational constraint.

Let  $\mathcal{D}$ ,  $\vec{V}$ ,  $\text{Dom}$  represent the data schema, set of session variables, and the data domain, respectively. Given a transducer  $T = (\mathcal{D}, \text{Dom}, \vec{V}, \vec{A})$ , we say a state of  $T$  is a database instance of  $\mathcal{D}$  and a valuation of all variables in  $\vec{V}$ . Given two states  $s$  and  $s'$ , we write  $s' \in T(s)$  if  $s'$  is the result of applying all the assignments of  $T$  on  $s$ , i.e.,  $T(s, s')$  evaluates to true. Note that there might be multiple states resulting from the same transition and input state. If  $s'$  is the only such post-state, we write  $s' = T(s)$ . We now define the notion of symbolic image computation.

**Definition 4.1** Let  $T = (\mathcal{D}, \text{Dom}, \vec{V}, \vec{A})$  be a path transducer and  $I$  is a relational constraint over  $\mathcal{D}$  and  $\vec{V}$ . The preimage  $\text{Pre}(T, I)$  is defined as  $\text{Pre}(T, I) = \{s \mid s' \in T(s) \wedge s' \in I\}$ , and the postimage is

$\text{Post}(T, I) = \{s' \mid s' \in T(s) \wedge s \in I\}$ . When  $I$  is true, we simply write  $\text{Pre}(T, I)$  and  $\text{Post}(T, I)$  as  $\text{Pre}(T)$  and  $\text{Post}(T)$ , respectively.  $\blacksquare$

The complexity of generating a relational constraint that represents  $\text{Pre}(T, S)$  is in polynomial time. This can be easily inferred from the fact that a transducer  $T$  is a collection of assignments on relations and variables. Simply replace all primed variable and relation by its RHS. Note that we do not have a similar result for post-image. We now proceed to a finite scope solution for the satisfiability problem. We illustrate the idea using an example.

<pre> 1  sig vchar {} 2  sig UserRecord{ 3    uname: vchar , 4    pwd: vchar 5  } 6 7  sig SessionRecord{ 8    sid: Int , 9    sname: vchar 10 } 11 12 sig MemberRecord{ 13   sid: Int , 14   uname: vchar 15 } 16 17 sig UserTable{ 18   list: set UserRecord 19 }{ 20   //primary key 21   all x,y: list   22     x.uname = y.uname =&gt; x=y 23 } 24 25 sig SessionTable{ 26   list: set SessionRecord 27 }{ 28   //primary key 29   all x,y: list   30     x.sid = y.sid =&gt; x=y 31 } </pre>	<pre> 32 sig MemberTable{ 33   list: set MemberRecord 34 }{ 35   //primary key ... and foreign key 1... 36   //foreign key 2 37   all x: list   one y: SessionTable   38     one z : y.list   z.sid = x.sid 39 } 40 41 fact aboutRecords{ 42   ( all x: UserRecord   43     one u: UserTable   x in u.list ) 44   ... 45 } 46 47 // The following are for query 48 one sig c_s1, c_s2 extends vchar {} 49 pred part1[d:vchar]{ 50   some a,c:Int   some b: vchar   51     a=c &amp;&amp; b in c_s1 &amp;&amp; 52     (some y: SessionTable   some x: y.list   53       x.sid=a &amp;&amp; x.sname=b) 54     &amp;&amp; (some y: MemberTable   some x: y.list   55       x.sid=c &amp;&amp; x.uname=d) 56 } 57 pred part2[d:vchar]{ 58   ... 59 } 60 pred query[d:vchar]{ 61   part1[d] and !part2[d] 62 } </pre>
--	--

Figure 2: Sample ALLOY Specification

**Example 4.2** Assume that we are interested in performing the following query: list all users who are members of session “s1” but not of session “s2”. This query can be expressed using the following relational algebra formula:

$$\pi_4(\sigma_{2=s1'}\sigma_{1=3}(S \times M)) - \pi_4(\sigma_{2=s2'}\sigma_{1=3}(S \times M))$$

The Alloy specification of the query is shown in Figure 2. The first part (lines 1 to 40) defines the data schema: relations and the corresponding primary key constraint for each relation. Each row of a relation is defined as a sig (data type) in Alloy, and each column is defined as an attribute. A relation is defined as a set of the corresponding rows (records). Constraints (such as primary and foreign keys) can be defined conveniently as first order relational logic formula. The aboutRecords fact asks Alloy to perform search on those records in a relation only.

The second part of the Alloy specification encodes the query. It essentially follows the idea of converting a relational algebra to a first order logic query. The predicate query encodes the desired difference operation. By running the query for exactly 1 instance of each data relation and 3 instances of data records for each relation, a database instance is generated in 78ms to satisfy the query, on a laptop PC with 4GM memory and 3GHz CPU. It is shown as below:

1. SessionTable:  $\{(vchar3, c\_s1)\}$
2. MemberTable:  $\{(vchar3, vchar2)\}$
3. UserTable:  $\{(vchar2, vchar3)\}$

Here all foreign key constraints are followed, e.g., the sid column (vchar3) of the only record of MemberTable is the same as the one in SessionTable.  $c\_s1$  denotes the constant  $s1$ , and it is the value of the sname of the SessionRecord. For simplicity, the uninterpreted function  $f$  (encryption) is not encoded in this example. It is available in the Alloy model for Step 5 in §5. ■

## 5 Call Sequence Synthesis

This section introduces the synthesis algorithm that generates a test case composed of a sequence of HTTP requests. The purpose is to reach a certain database state for achieving coverage goals. We begin with some formal definitions needed for discussions later.

**Definition 5.1** An HTTP request  $r$  is a tuple  $(u, \vec{p})$  where  $u$  is the base URL, and  $\vec{p}$  is a finite set of parameters. Each parameter is a tuple  $(n, v)$  where  $n$  is the name of the parameter and  $v$  is its value.  $\vec{p}[n]$  denotes the value of parameter  $n$ . ■

**Definition 5.2** A call sequence is a finite sequence of HTTP requests  $r_1, \dots, r_n$ . We call the corresponding HTTP responses  $s_1, \dots, s_n$ . Each  $s_i$  is a string that represents the contents of the HTML file returned. ■

A call sequence is also written as  $(r_1, s_1), (r_2, s_2), \dots, (r_n, s_n)$  when HTTP responses need to be considered. We call a response *bad*, if it contains error messages such as HTTP 505 internal error.

### 5.1 Call Sequence Synthesis Algorithm

Figure 3 presents a heuristic algorithm that synthesizes a call sequence of web servlets. Its input includes (1) initial database states specified using a relational constraint  $S_0$ , (2) a finite set of transducers  $\mathcal{T}$ , and (3) the objective states represented using relational constraint  $\phi$ .

As shown in Figure 3, the backtracking algorithm attempts to simulate the changes of system states, using pre-image computations. Here  $S$  is a stack which stores the call sequence. Each element of  $S$  is a tuple which records the current servlet being attempted and the corresponding system states (represented using a relational constraint). Every iteration of the backtracking loop tries to identify a new servlet which modifies the system state, geared towards the direction of initial states  $S_0$ . The current system states (represented using  $\phi$ ) are compared with  $S_0$  using function `getModified`. It tries to extract the set of variables and relations whose “value domain” change between the two sets of symbolic states. The value domain of a variable  $v$  in constraint  $\phi_1$  is formally defined using formula  $\phi_1(v) = \{a \mid s \in \phi_1 \wedge s[v] = a\}$ . Here  $s[v]$  is the value of variable  $v$  in state  $s$ . Clearly  $\phi_1(v)$  can be computed using existential elimination of all other variables/relations in the formula of  $\phi_1$ . If a finite scope is given, this can be achieved using first order relational logic in Alloy [17]. Once the “difference” of the two symbolic constraints  $\phi_1$  and  $\phi_2$  is found, each servlet is statically examined. A servlet that modifies some variables/relations in  $M_1 - M_2$

is identified and pushed to stack  $S$ . If none is found, the search process backtracks. It proceeds until the initial state constraint  $S_0$  has some intersection with the current system state; or the algorithm fails and returns an empty stack.

We use a case study to illustrate the effectiveness of the algorithm. Applying the algorithm to real-world examples remains as our future work. The heuristics works better with INSERT statements than UPDATE statements.

```

1 //S0: desired initial states,  $\phi$ : desired path condition,  $\mathcal{T}$ : a finite set of path transducers
2 Procedure CallSeqGen( $\mathcal{T}$ ,  $S_0$ ,  $\phi$ )
3   Let  $S = [(\perp, \phi)]$ 
4   //S is a stack which stores the intended call sequence to return
5   //Each element of S is a tuple of (action, current state)
6   while( $S_0 \cap \phi = \emptyset$  and  $|S| \geq 1$ ) {
7     Let  $\phi_1$  and  $\phi_2$  be the states stored in the top 2 elements of S. If  $|S| = 1$ , let  $\phi_2$  be true
8     Let  $M_1 = \text{getModified}(\phi_1, S_0)$ 
9     Let  $M_2 = \text{getModified}(\phi_2, S_0)$ 
10    Find a path transducer  $s \in \mathcal{T}$  which modifies some target  $v \in M_1 - M_2$  and  $\text{Pre}(s, \phi) \neq \text{false}$ 
11    If no servlet is available then backtrack:
12      remove the first element of S; continue
13     $\phi := \text{Pre}(s, \phi)$ ;  $S := (s, \phi) \circ S$ 
14  }
15 return S

16 //return a set of session variables and relations that are changed between the two sets of states represented by  $\phi_1$  and  $\phi_2$ 
17 Procedure getModified( $\phi_1, \phi_2$ )
18 Let  $R = \{\}$  // R is the set of variables and relations to return
19 foreach  $v \in V \cup \mathcal{D}$ : //v could be a session variable or relation
20   Let  $\phi_1(v) = \{a \mid s \in \phi_1 \wedge s[v] = a\}$  //here s is a state belong to  $\phi_1$  and  $s[v]$  is the value of v in s
21   Let  $\phi_2(v) = \{a \mid s \in \phi_2 \wedge s[v] = a\}$ 
22   if  $\phi_1(v) - \phi_2(v) \neq \emptyset$  or  $\phi_2(v) - \phi_1(v) \neq \emptyset$ :  $R = R + \{v\}$ 
23 return R

```

Figure 3: Call Sequence Generation Algorithm

## 5.2 Case Study

The coverage goal is to reach line 45 in Listing 1. We would like to synthesize a call sequence that reaches the line. The initial state  $S_0$  is expressed using formula  $\#u = \text{null} \wedge S = M = U = \emptyset$ , i.e., the database is empty and none of the global session variables is set. The target constraint  $c_0$  is represented by relational constraint  $\#u \neq \text{null} \wedge \text{SAT}(\sigma_{2=\#u}M)$ , i.e., table  $M$  has one record whose second attribute “uname” has the value  $\#u$  (the user name contained in session variable  $\#uname$ ). This is essentially the pre-condition of a path transducer of `Showsessions.php` that reaches line 45.

**Step 1:** We start with the path transducer of `Showsessions.php`. The first step is to compute the pre-image of the transducer.

$$\text{Pre}(\#u \neq \text{null} \wedge \text{SAT}(\sigma_{2=\#u}M) \wedge S' = S \wedge U' = U \wedge M' = M \wedge \#u' = \#u, \text{true}) \quad (6)$$

Recall that `Pre` takes two parameters: (1) the transition relation, and (2) the post image. The algorithm of `Pre` is straightforward. First, convert all “current state” variables in the post condition (i.e., the second parameter of `Pre`) to “next state”, construct the conjunction of the two parameters, and then replace each post state variable (e.g., “ $S'$ ”) with its RHS (e.g., “ $S$ ”) in the assignment of transition relation. We then obtain the precondition. Notice that this algorithm is based on the assumption that in each assignment, post-state variables occur in LHS only. Clearly, for step 1, the resulting pre-image is shown below:



$$N_4 : \#u \neq \text{null} \wedge \text{SAT}(\sigma_{2=\#u}M)$$

The synthesis algorithm cannot terminate here, because the goal  $S_0 \cap N_4 \neq \emptyset$  is not met. We need to determine which is the action before `Showsessions.php`. The candidates are `Login.php` which resets  $\#u$  and `Addmember.php` which updates  $M$ . Without loss of generality, we choose `Login.php` as the preceding action of `Showsessions.php`.

**Step 2:** The current call sequence is LS (`Login.php` and then `Showsessions.php`). The post condition of L is  $N_4$ . We can compute its pre-image as follows.

$$\begin{aligned} \text{Pre}( & (\$u_L, f(\$p_L)) \in U \wedge \#u = \text{null} \wedge \#u' = \$u_L \wedge U' = U \wedge S' = S \wedge M' = M \wedge \\ & \#u' \neq \text{null} \wedge \text{SAT}(\sigma_{2=\#u'}M')) \end{aligned}$$

Here the first row is the transition rule for `Login.php`, and the second row is the primed form of  $N_4$ . We use the one parameter version of the `Pre` function here, and the post-image has to be primed. The same algorithm results in the following pre-image. Note that here  $\$u_L$  and  $\$p_L$  are the input parameters (uname and pwd) of `Login.php`.

$$N_3 : (\$u_L, f(\$p_L)) \in U \wedge \#u = \text{null} \wedge \$u_L \neq \text{null} \wedge \text{SAT}(\sigma_{2=\$u_L}M)$$

**Step 3:** Next, `Addmember.php` is identified by the heuristic algorithm as the preceding action. The pre-image is computed as below. As usual, the first row is the transition rule and the second row is the primed form of  $N_3$ . Notice that  $\$u_L$  is the input parameter for `Login.php`. It is treated as a symbolic literal (its value cannot change during the execution). We do not have to “prime” it in the post-image.

$$\begin{aligned} \text{Pre}( & M' = M \cup (\pi_1(\sigma_{2=\$s_A}(S)) \times \{(\$u_A)\}) \wedge U' = U \wedge S' = S \wedge \#u' = \#u \wedge \\ & (\$u_L, f(\$p_L)) \in U' \wedge \#u' = \text{null} \wedge \$u_L \neq \text{null} \wedge \text{SAT}(\sigma_{2=\$u_L}M')) \end{aligned}$$

Here  $\$u_A$  and  $\$s_A$  are the input parameters (uname and session name) of the `Addmember.php`. The pre-image computation results in the following state constraint:

$$\text{SAT}(\sigma_{2=\$u_L}(M \cup (\pi_1(\sigma_{2=\$s_A}(S)) \times \{(\$u_A)\}))) \wedge (\$u_L, f(\$p_L)) \in U \wedge \#u = \text{null} \wedge \$u_L \neq \text{null}$$

**Step 4:** The pre-image computation is listed as below for `Generaloptions.php` (to add a user). Here  $\$u_G$  and  $\$p_G$  are the input parameters (uname and pwd).

$$\begin{aligned} \text{Pre}( & U' = U \cup \{(\$u_G, f(\$p_G))\} \wedge r(\$p_G) \wedge M' = M \wedge S' = S \wedge \#u' = \#u \wedge \\ & \text{SAT}(\sigma_{2=\$u_L}(M' \cup (\pi_1(\sigma_{2=\$s_A}(S')) \times \{(\$u_A)\}))) \wedge (\$u_L, f(\$p_L)) \in U' \wedge \#u' = \text{null} \wedge \$u_L \neq \text{null} \end{aligned}$$

It results in the following:

$$N_1 : r(\$p_G) \wedge \text{SAT}(\sigma_{2=\$u_L}(M \cup (\pi_1(\sigma_{2=\$s_A}(S)) \times \{(\$u_A)\}))) \wedge (\$u_L, f(\$p_L)) \in U \cup \{(\$u_G, f(\$p_G))\} \wedge \#u = \text{null} \wedge \$u_L \neq \text{null}$$

Since  $S_0 \cap N_1 = \emptyset$ , The heuristic algorithm has to proceed to modify table S.

**Step 5:** Using constraint  $N_1$  as the post condition of `Insertsession.php`, we have:

$$\text{Pre} ( S' = S \cup \{(|S|+1, \$s_I)\} \wedge M' = M \wedge U' = U \wedge \#u' = \#u \wedge \\ r(\$p_G) \wedge \text{SAT}(\sigma_{2=\$u_L}(M' \cup (\pi_1(\sigma_{2=\$s_A}(S')) \times \{(\$u_A)\})) \wedge (\$u_L, f(\$p_L)) \in U' \cup \{(\$u_G, f(\$p_G))\} \wedge \#u' = \text{null} \wedge \$u_L \neq \text{null})$$

This leads to the final constraint  $N_0$  :

$$N_0 : r(\$p_G) \wedge \text{SAT}(\sigma_{2=\$u_L}(M \cup (\pi_1(\sigma_{2=\$s_A}(S \cup \{(|S|+1, \$s_I)\})) \times \{(\$u_A)\})) \wedge (\$u_L, f(\$p_L)) \in U \cup \{(\$u_G, f(\$p_G))\} \wedge \#u = \text{null} \wedge \$u_L \neq \text{null}$$

If we test  $S_0 \cap N_0$ , the following is an assignment which provides satisfiability:

$$M = S = U = \emptyset \wedge \$u_L = \$u_G = \$u_A = a \wedge \$p_G = \$p_L = b \wedge \$s_I = \$s_A = c \wedge \#u = \text{null}$$

Here  $a, b, c$  are three constants generated by the model finder. The constraint  $r(b)$  can be discharged separately using a string constraint solver like [27] and [13]. When the string constraint  $r(b)$  is ignored, Alloy Analyzer is able to generate the model in 1.07 seconds and solve the model in 57ms. In the model generated by Alloy, constants  $a$  is equal to  $c$ , and the encryption function is properly modeled in Alloy and it has the property:  $\forall a, b : f(a) = f(b) \Leftrightarrow a = b$ .

## 6 Detecting Workflow Attack

This section briefly discusses the extension of the algorithm for detecting workflow attacks [3]. Since web servlets are openly accessible to end users, hackers could potentially access the web application and violate its intended “workflow” logic (e.g., shipping a product before payment is handled). Such an attack can cause great financial losses. To model the “intended” workflow of a web application, we could apply string analysis like [6] and collect the URLs that are contained in the HTML generated by a servlet. Then, a workflow attack can be defined formally.

**Definition 6.1** An enhanced path transducer is a tuple  $T = (\mathcal{D}, \text{Dom}, \vec{V}, \vec{A}, \vec{L}, U)$  where  $\mathcal{D}$ ,  $\text{Dom}$ ,  $\vec{V}$ ,  $\vec{A}$  are as defined in Definition 3.1.  $\vec{L}$  is a set of transducers that  $T$  can navigate to.  $U$  is the base URL where the corresponding servlet is deployed. ■

Given a collection of path transducers, we use  $\vec{L}(U)$  to denote the union of the  $\vec{L}$  components of all path transducers who have  $U$  as the base URL.

**Definition 6.2** Given a web application that consists of  $\mathcal{T}$ , a finite set of path transducers, a workflow attack is a call sequence  $(r_1, s_1), \dots, (r_n, s_n)$  where none of the responses is bad and there exists  $i \in [1, n-1]$  s.t.  $r_{i+1}[0] \notin \vec{L}(r_i[0])$ . Here  $r[0]$  to refer to the first element, i.e., the request URL, of a request  $r$ . ■

The `CallSeqGen` algorithm in Figure 3 can be slightly modified to discover workflow attacks. The inputs to the algorithm are: (1)  $\mathcal{T}$ , the collection of enhanced path transducers; (2)  $S_0 : \forall v \in V \ v = \text{null}$ , the initial state where all session variables are null (database is not necessary to be empty); and (3) `true` as the desired post condition. Then at line 10 of Figure 3, append an additional condition shown as below when selecting  $s$ :

$$|S| = 2 \Rightarrow s' \notin \vec{L}(s)$$

where  $s$  is the path transducer to be selected,  $s'$  is the second top element currently in the stack, and  $\vec{L}(s)$  is the  $\vec{L}$  component of  $s$ . Enforcing  $|S| = 2$  is to find the shortest attack string, such that the violation action is the last one in the sequence. Note that due to the backtracking nature of the algorithm, if the current guess of  $s$  does not work, the algorithm will trace back to find another candidate.

## 7 Related Work and Conclusion

This work is closely related to testing database applications, e.g., code coverage and database unit testing [9, 12, 5]. Recently, query aware database generators are reported to significantly improve the size and quality of test cases (see [20, 4]). Our work goes one more step beyond database generation – we synthesize the call sequence of web application servlets.

The general satisfiability problem for relational database is undecidable [1]. In practice, one has to adopt either approximation or solving decidable fragments. Emmi, Majumdar, and Sen generate database input for programs in [12], and a decidable fragment of SQL is handled, which does not allow join and negation. In [4], a reverse query processing technique is developed, which introduces approximation when handling negation. In [20], Khalek *et al.* translate SQL queries to relational logic formula, and use Alloy Analyzer [17] to perform model finding. We adopt a similar approach to that of Khalek's.

Extracting the interface of a web servlet has been investigated by Halfond and Orso [14, 15], where static analysis is used to identify the parameters accepted by a web application. In [16], they went a step further to collect path conditions as web servlet interfaces. In this paper, the interface extraction concentrates on the manipulation of backend databases. Each web application servlet is modeled as a set of path transducers, which is inspired by the relational transducer model [2] introduced by Abiteboul *et al.* for modeling electronic commerce. Automated verification of relational transducers is discussed in [23, 10]. The problem is in general undecidable.

Testing web applications has its unique challenges. Due to the existence of server states, e.g., session variables and HTTP cookies, a test case of web applications usually consists of a sequence of HTTP requests. This is often called session based testing [22, 11, 24]. In the aforementioned work, HTTP sessions are either manually created or collected by parsing Apache server log. The technique presented in this paper synthesizes test cases automatically. It has the potential to improve code coverage.

We have outlined a framework for automatically generating test cases of web applications. The key idea is to first extract the interface of each web servlet as a set of single path relational transducers. Then we could solve symbolic constraints on relational databases and synthesize a call sequence of web servlets. Our future work includes the implementation of the proposed technique and investigation of more efficient constraint solving techniques.

## References

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61:236–269, 2000.
- [3] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS07)*, pages 25–35, 2007.
- [4] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proceedings of IEEE 23rd International Conference on Data Engineering*, pages 506–515, 2007.
- [5] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st international conference on Very large data bases (VLDB)*, pages 1097–1107, 2005.
- [6] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from <http://www.brics.dk/JSA/>.

- [7] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, Queensland, Australia, September 5–7, 2007.
- [8] DBUnit. <http://www.dbunit.org/>.
- [9] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *Proceedings of the 27th international conference on Software engineering (ICSE)*, pages 78–87, 2005.
- [10] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 71–82, 2004.
- [11] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.
- [12] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*, pages 151–162, 2007.
- [13] X. Fu and C.C. Li. Modeling regular replacement for string constraint solving. In *Proc. of the 2nd NASA Formal Methods Symposium*, pages 67–76, 2010.
- [14] W. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, Dubrovnik, Croatia, September 2007.
- [15] W. Halfond and A. Orso. Automated Identification of Parameter Mismatches in Web Applications. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2008)*, pages 181–191, Atlanta, Georgia, USA, November 2008.
- [16] William G. J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA*, pages 285–296, 2009.
- [17] D. Jackson. Alloy 3 Reference Manual. <http://alloy.mit.edu/reference-manual.pdf>.
- [18] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 130–139, 2000.
- [19] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.
- [20] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 238–247, 2008.
- [21] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [22] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings of the 19th IEEE international conference on automated software engineering*, pages 132–141, 2004.
- [23] Marc Spielmann. Verification of relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 66(1):40–65, 2003.
- [24] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE)*, pages 253–262, 2005.
- [25] SQLUnit. <http://sqlunit.sourceforge.net/>.
- [26] Paul Tarjan and Nick McKeown. Stanford conference and research forum. <http://scarf.sourceforge.net>, 2006.
- [27] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. of the 15th SPIN Workshop on Model Checking Software*, pages 306–324, 2008.